



Computer and Computational Sciences Division

CCS-4:Transport Methods Group

To/MS: Distribution
From/MS: Todd J. Urbatsch/CCS-4 D409
Phone/FAX: (505)667-3513
Symbol: CCS-4:03-21(U) (LA-UR-03-3931)
Date: June 11, 2003

Subject: Contributions to the FY03 Code Strategy

Executive Summary

This memo contains my contributions to the Code Strategy Implementation Plan, which was begun in summer 2002 by the Code Strategy Implementation Team (CSIT), of which I was a member. These contributions are in the form of three sections, each with an executive summary:

- Numerical Methods, Algorithms, and Component Delivery
- Components for the ASC Program
- Development Environments (DE) for the ASC Program

The first two sections were completed in October, 2002, and the third section was completed in December, 2002. These contributions are being published 6–8 months after the fact because the Code Strategy Implementation Plan has not been disseminated widely, and I need to inform my group and my management what I contributed. Time has rendered some of the specifics of this writeup outdated, but the ideas and proposed strategy remain relevant.

The viewpoints expressed here are mainly due to my experience with the Jayenne IMC Project in CCS-4 (Transport Methods), but are meant to represent the view of methods and ASC software developers in CCS Division. The CCS-4 group members, the Components Project Leaders, and other members of the CSIT commented on these writeups before they were submitted; all comments were incorporated into the writeup.

The main implementation details to be taken from these writeups are that all new code development—even on legacy codes—be done as components and that the program to move to a component-based development environment.

1. Numerical Methods, Algorithms, and Component Delivery

1.1. Executive Summary

The validated software used for LANL's mission must be comprised of verified physics components that contain the best numerical methods and algorithms. Using components to build software is a foundational development practice that makes verification tractable. Verification is the assurance that equations are solved correctly and the assurance that software behaves as expected. Unless software behaves as expected, validation of the software is pointless, and predictive capability is impossible. Components, whether developed by component teams or ASC application code teams,

will be contributed to the ASC application codes via industry-standard processes for staged delivery of high-quality software. The metrics for judging the quality of a component are classed into physics, numerics, and software categories. In general, the component must behave as expected. The most important metric for a component is its testability. Other metrics are interoperability and degree of efficiency at which LANL’s resources are utilized. Information for metric assessment and decision points will be procured through various levels of peer review, up to ASC-wide. ASC management will make or approve the final assessment and decisions.

1.2. Introduction

The validated software used for LANL’s mission must be comprised of verified physics components that contain the best numerical methods and algorithms. Using components to build software is a foundational development practice that makes verification tractable. Verification is the assurance that equations are solved correctly and the assurance that software behaves as expected. Unless software behaves as expected, validation of the software is pointless, and predictive capability is impossible.

Staged delivery of high-quality components to the ASC application codes will occur via mature, documented, industry-standard processes. One example is the “unified process” [1]. These expectations apply to whomever develops components, whether it is a component team or an ASC application code team.

Briefly, a software component has a well-defined interface (input and output), with well-documented, repeatable functionality. Moreover, aside from interface code (also known as “glue” or “middleware” code), a component does not change when deployed to different application codes. Components are necessary for other SQE practices, such as unit testing and levelized design. Components and unit testing make debugging and verification tractable because the work associated with unit testing scales algorithmically, not geometrically as it does with integral testing alone. Component-based software development helps locate bugs quickly and assign proper responsibility. Component-based software development makes it easier to test new methods and may help bridge the gap between research and code development.

Therefore, the objective of methods research and component development is to deliver these high-quality components for use in the ASC validation efforts. We outline the metrics of a successful component and describe the peer-review for obtaining information for ASC management to assess how well a component meets these metrics.

1.3. Iterative Steps in Component Development

The iterative steps in developing a component are inception, design, construction, transition.

1.3.1. Inception. Specified requirements are a component’s seed of inception. For a high-level physics component, the requirements are identified and prioritized by ASC management via user requests and negotiations between pertinent stakeholders. Pertinent stakeholders include component developers, application code developers, users, designers, program managers, group managers, and division managers. Component requirements must balance physics, methods, algorithms, com-

puter hardware, environments, experimental requirements and data, etc. Sub-components should be shared between high-level physics components when appropriate. When negotiations fail and ASC management needs additional input, it may be necessary to convene the Code Strategy Implementation Team or a panel of Science Advisors from the directorate.

1.3.2. Design. Component developers are responsible for writing up the specifications of the component. The document should detail what they will deliver and the behavior of the software, given their understanding of the requirements. This document should also provide a schedule so that progress can be tracked by developers, customers, and management. The input, output, and behavior of a component is driven by contractual requirements. Interfaces are negotiated, agreed upon, and documented. The component developer must document the assumptions and reasons behind the choice of methods, algorithms, and design.

Component developers are also responsible for documenting the component’s numerical equations, the continuum equations, and quantification of numerical errors. Merely referencing journal articles is not enough. These documents are to be living documents, ideally stored with the source code, that represent ongoing changes in the physics, numerics, and code.

The design documentation will also serve as material for peer review of the design.

1.3.3. Construction. Components must be constructed using standard languages, without pushing the language envelope or over-stressing compilers. Component developers are responsible for documenting the methods used by the component, commenting the component, testing the component, getting the component peer-reviewed (e.g., code reviews, code walk-throughs, and pair programming), and documenting the component and its interface. By the same token, component developers are responsible for peer-reviewing other components. The component must be verified to behave as expected (e.g., Design-by-Contract(tm), error analysis, accuracy, convergence, stability, etc.).

1.3.4. Transition. The transition of a component is the redesign, rewriting, or refactoring of the component necessary to improve the component or to address the inevitable changes in systems, hardware, related software, interfaces, and requirements. Interfaces are designed to be static, but will necessarily change, and therefore should be managed to change in a controlled fashion.

1.4. Delivering a Component to an ASC Application

When the first staged-delivery of component is released and ready for integration, both the component team and ASC application code team share the responsibility for integration. Both teams are responsible for the multi-component issues that will arise. Thus, component teams share responsibility for how their component interacts with other components, how their components are used, and user support. Component developers may assist users or take part in validation efforts. Responsibility for component maintenance should motivate use of the best SQE practices to develop high-quality software.

Responsibility for a component beyond its development must be enforced. This extended and overlapping responsibility is necessary because of LANL’s limited-scope, focused mission, and it is

expected because of the corresponding funding levels and the limited staffing levels.

1.5. Resource Allocation

To balance independent research, directed research, and programmatic delivery, activities that deliver capability to the ASC application codes by 2005 or before must have higher priority. Therefore, independent research that has little or no bearing on programmatic activities must have the lowest priority. It is understood that verification and undergoing and performing peer reviews are inherent costs of software development.

1.6. Metrics for Components

The metrics for judging the quality of a component are classed into physics, numerics, and software categories. If a given release, or delivery, of a component does not meet the specified requirements, the probability that future deliveries will meet the requirements must be ascertained. For each category, we list pertinent questions.

1.6.1. *Physics.* The physics requirements are stipulated and documented. Does the component meet the stipulated and documented physics requirements? Have the physics requirements changed? Does the documentation reflect these changes?

1.6.2. *Numerics.* Do results from the component match analytic results? How well, quantitatively? Does error analysis of the component match the expected analytic analysis? If not, why not?

1.6.3. *Software Engineering.* Is the software readable? Is the style consistent? Is the design understandable? Is there a sufficient amount of comments?

How easy is it to verify the component? Is a component broken into smaller, easily testable, sub-components? If not, why not? Does each sub-component have a unit test? To what degree is the component verified in the way of unit tests? Are the unit tests used for regression testing?

What is the bug history? How long did it take to make the first delivery? How long has or will it take to make the next delivery?

How many application codes are using the component? Does the component have a peer with which it is interchanged? Compare and contrast the peers. For a potentially reusable component, does its generality come at the cost of reliability, maintainability, and development time?

What are the building, compiling, and runtime qualities of the component? Is the time to build and compile tolerable? Is the component's efficiency sufficient? Has it been checked for memory leaks? How seamless is the component integration? What are the costs and benefits of the component?

1.7. Methods for Assessment of Metrics Satisfaction

Peer-reviews and customer satisfaction will be used to assess whether a component has met its stipulated metrics.

Those who specified the physics requirements must agree that the requirements were met, or they must agree to modifications of the requirements. The agreements must be documented for everyone's benefit.

Peer-review will be used to critique the numerics associated with the component. Personnel from within the component developers' group or specialty, from other component teams, and customers will assess the choice of numerical methods and algorithms, the assumptions behind those choices, the computational efficiency of the methods and algorithms chosen, and the errors and quantification of errors in the numerical methods as coded.

Peer-review will be used also to critique the software engineering aspects of the component. Personnel from within the component developers' group, from other component teams, and from software engineering groups will be tasked to assess the readability, testability, maintainability, ability to handle expected requirements, efficiency, and overall usefulness (amount of testing, sharing, and interchanging) as a component. Once a component has been delivered to the end user, testing should have eliminated most of the software bugs. After delivery, the quality of the component will be questioned insofar as the number and frequency of bugs, and who found the bugs. Part of this peer-review will be live demonstrations of the version control, building, and verification of the software.

Both undergoing and performing peer-review must be an inherent part of component developers' business. They must be funded as such.

1.8. Component Decision Points

Information from peer-reviews will be used for making technical decisions regarding the component. When a component meets the testability criteria, it contributes directly to an improved predictive capability and should continue its existence. When a component is being used in an ASC applications code and its testability and quality outweigh any of its performance inefficiencies, the component is a success. Sharing a high-quality, well-tested component simply increases its importance. The ability to interchange a high-quality, well-tested component with another component indicates an effective use of components. On the other hand, although the reusability of a specific component across many applications is desired, reusability should not dictate the software design to the serious detriment of reliability, maintainability, and development time. If component-based software development gets more ASC-funded personnel involved in LANL's mission, it is an even greater success.

2. Components for the ASC Program

2.1. Executive Summary

One of the goals of the Code Strategy is to increase the level of component-based software development in the ASC code effort. The effect of component-based software development is improved verification and better utilization of LANL's resources. The goal of the Code Strategy Implementation Plan is to change the atmosphere of unnecessary competition and poor verification to an

atmosphere of collaboration in developing well-verified software. Thus, LANL code developers must now consider whether the capability they are developing or planning to develop is suitable to be developed as a component and whether or not multiple application codes can use the component. The ASC sub-project 1.1.2.3.4: Component Definition and Architecture R&D is intended to be a seed and catalyst for implementing this change in software development practice. Its plan is to construct, help construct, or facilitate the construction of new or converted components and to integrate, help integrate, or facilitate the integration of both new, ongoing, and existing components. Quarterly ASC-wide gatherings and peer review will be the forum for requesting and prioritizing components. ASC-level management, with involved division and group management, must decide upon and approve the recommended or suggested prioritizations. Recommendations and decision must be documented.

2.2. Introduction

One of the goals of the Code Strategy is to increase the level of component-based software development in the ASC code effort. Using components to develop software is one of many useful SQE practices [2]. Components can be individually tested; these unit tests aid in the development, maintenance, and extension of the software. Moreover, components can be arranged in a hierarchical, leveled design to build up higher-level components. Unit testing makes predictive capability approachable, because errors can be nearly impossible to detect from the highest, executable level alone. Well-tested, verified code frees up the users to concentrate on the validation of the physical models in the software.

Beyond testing, components can be shared and interchanged. This interoperability allows code re-use, increases capability, and reduces duplicated work, all of which will tend to further improve the overall quality of ASC software and increase the efficiency of ASC software development. Components allow true method-to-method comparisons when those methods reside in different codes. Component histories, both successes and failures, will be kept in a source code repository for future reference.

Although this Code Strategy Implementation Plan specifies that all codes—both legacy and ASC—continue to live, components will help manage the number of codes over time. The transfer of technology from a legacy code to an ASC code can occur by converting the legacy capability to a component, using that component in the legacy code, and using that component in ASC codes. Legacy codes can be retired legitimately when ASC codes contain a sufficient amount of their capability.

2.3. Plan

The general plan is to construct, help construct, or facilitate the construction of new or converted components and to integrate, help integrate, or facilitate the integration of both new, ongoing, and existing components.

2.3.1. *Physics.* Identify, or locate and collate, the continuum and discrete equations used in the ASC application codes.

2.3.2. *Design.* Construct conceptual and actual leveled design diagrams of the components in the ASC software program. This tool will help locate places where components make sense.

2.3.3. *Requests and Prioritization of Components.* ASC-wide peer review will be the forum for requesting and prioritizing components. ASC-level management, with input from involved division and group management, must decide upon and approve the recommended or suggested prioritizations.

When an ASC application code team plans a new capability, they must now consider developing this capability as a component. They must query the other code teams about sharing the component and design the component interface accordingly. Peer reviews of different levels will focus on code design, verification, and validation.

2.3.4. *Interlanguage Issues.* The standard languages F90 and C++ communicate effectively now through flat interfaces. How can we make these interfaces more robust and efficient? Will we be able to migrate our components to more ambitious, language-independent component models, such as the Common Component Architecture (CCA)? We will demonstrate the effectiveness of coupling with other languages, such as Java—in which the CarteBlanca code is written, with F90 and C++.

2.3.5. *Proposed Component and Component Integration Efforts.* Below are several suggested component efforts that are ripe for investigation, development, and/or utilization.

-materials package

X-7 has a materials package for Lagrangian hydrodynamics. T-14 has a materials package for analytic models and a design more appropriate for ALE/Eulerian hydrodynamics. Help X-7 design for ALE, and facilitate the introduction of EOSPAC into T-14's package. Identify and extract components common in the two packages. Seek involvement of others.

-EOSPAC, NDI, Gandolf, CDI, etc.

Encourage and assist all codes to use these existing component. Is there a legitimate reason for codes not to use these components? If so, how can that be remedied? Can users of these components contribute to the development, testing, and extension of them?

-production/depletion

Many code-pirated (code that is copied and tweaked “slightly”) versions exist. Can the leading, most modern effort be componentized and shared?

-tracers

Componentize existing capabilities so they can be tested and verified, then validated by the users, who may wish to extend the physics capability.

-hydro

Some efforts have already shown success with componentized hydro. Research and construct dimensionally unsplit, multi-material, Cartesian AMR Godunov hydrodynamics package, which will

fill in a small gap and which will serve as a testbed for unsplit radiation-hydrodynamics. This hydro component will target PSP with a goal of sharing components with the AMR hydro used in the Blanca Project. Consider also a tetrahedral-based hydrodynamics component.

-sub-zonal physics

The potential for componentizing sub-zonal physics and numerics should be investigated. E.g., interface tracking.

-transport

Transport packages already exist. Continue their development and integration as necessary. CCS-4 is beginning a deterministic thermal radiative transfer project that will produce diffusion and first-order discrete ordinates transport packages on unstructured meshes and that initially will target PSP.

-linear/nonlinear solvers

In-house and vendor libraries exist. Much of the solvers technology is inseparable from the numerical physics. Where should the line be drawn? A general nonlinear solver package should also be developed.

-mesh

Identify existing meshes and share. Construct a Cartesian AMR mesh type for PSP that can be used by the Blanca project's AMR efforts.

-setup

Componentize some of the X-8 efforts so other unstructured mesh-based codes can use their technology. Can tools such as ICEM be used in ASC codes other than PSP?

-remapping

Componentizing new and ongoing remapping efforts in X-8 and T-7 will allow physics components on different mesh types and variable locations to interact. Thus, it will be possible to take first looks at new methods without converting them to specific mesh types or discretizations. Implicit remapping that is tightly coupled to the physics may not be as independent as explicit remapping, but implicit and explicit remapping components should still be able to share sub-components.

3. Development Environments (DE) for the ASC Program

3.1. Executive Summary

A development environment (DE) is the policy and set of tools and components that allow software developers to write, compile, debug, document, and test software. The span of a DE varies. Some DEs are appropriately used across the freeware world, an entire institution, or a code team. At the broadest level, a DE is an environment where people can develop and share components.

What we really want to push is the DE that is an environment where people can develop and

share components. This collaborative, synergistic environment will be facilitated through regular ASC-wide gatherings, web postings, a common repository, and multiple levels of peer-review, the recommendations from which will be affirmed by ASC management. Whether component-based or not, we initially consider each ASC and legacy code as its own DE that continues to live. As the ASC software effort utilizes an increasing number of components—whether new, or harvested and converted—codes can share these components, and redundant codes can be retired legitimately.

3.2. Definition of Development Environment

A development environment (DE) is the policy and set of tools and components that allow software developers to write, compile, debug, document, and test software.

The span of a DE varies. Some DEs are appropriately used across the freeware world, an entire institution, or a code team. It follows that there exists a hierarchy of DEs. Thus, a code team would most likely interact with DEs at levels both above and below their own DE. The purpose of some DEs may be to provide components to a higher level DE; for others it may be to dictate how lower level components are to interact.

At the broadest level, a DE is an environment where people can develop and share components.

3.3. Examples of Development Environments

The current ASC Problem Solving Environment (PSE) already has or sponsors official tools, for example, tools for debugging (Totalview) and visualization (Ensign). The ASC PSE also concerns itself with platform dependent issues regarding, for example, standard libraries, vendors, and compilers.

An example of a DE is one that dictates how developers can write code or link components to produce an executable of a design code. Most of LANL's ASC and legacy codes fall into this category. The CartaBlanca code is a LANL example, written in Java, for methods research and delivery to outside customers.

At another level, Development Environments are a set of available components that can be used (and re-used) by higher level components and application codes. At LANL, for example, the Transport Methods Group has Draco, a library of unit-tested, re-usable radiation transport components.

Two efforts at Sandia National Laboratory, Sierra and Nevada, facilitate the development of components and dictates how they interact.

3.4. Implementation Plan for Investigating Development Environments

We will compare, contrast, and utilize LANL and non-LANL DEs as necessary.

There exists a language-independent component model called the Common Component Architecture (CCA). CCA has standards for component interfaces and other component service that may or may not be suitable to LANL's ASC effort. We will investigate how much, if any, of CCA standards and capability we should employ.

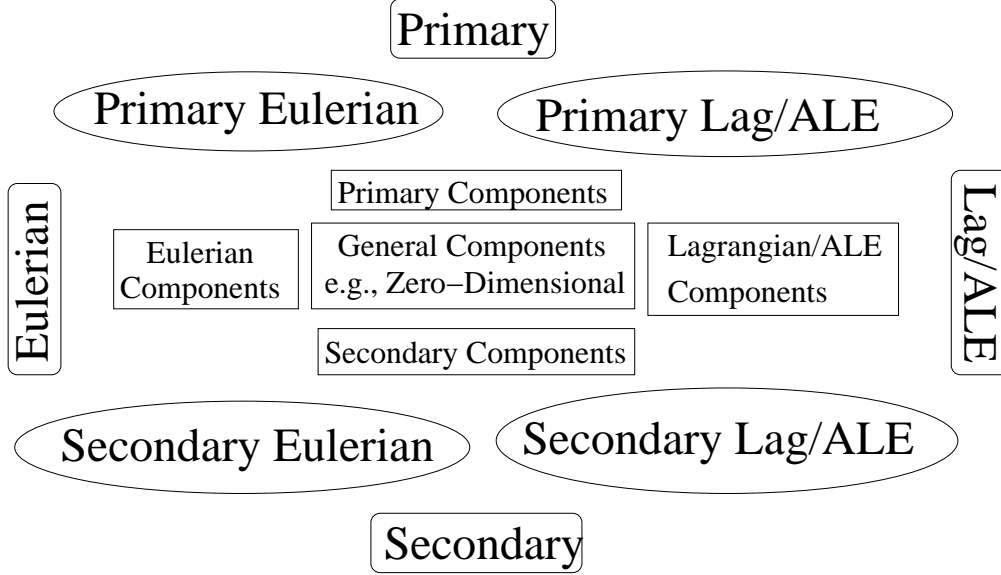


FIG. 1: Diagram of layered, general components that share centralized sub-components.

3.5. Implementation Plan for a LANL ASC-wide Development Environment

At the broadest level, a Development Environment is an environment where people can develop and share components. This sense of the DE is missing at LANL. We desire that the LANL ASC effort look and operate like a project with one mission. Such a collaborative, synergistic environment will be facilitated through regular ASC-wide gatherings, web postings, a common repository, and multiple levels of peer-review, the recommendations from which will be affirmed by ASC management.

Whether component-based or not, we initially consider each ASC and legacy code as its own DE that continues to live. However, it is understood that the Code Strategy specifies that any planned new capability, whether in an ASC or legacy code, or whether new or retrofitted, must be considered for development as a component, potentially sharable. If a legacy code has a capability that is required in an ASC code, that capability can be rewritten as a component, used and tested as a component in the legacy code, and then used in the ASC code(s).

As the ASC software effort utilizes an increasing number of components—whether new, or harvested and converted—codes can share these components, as shown in Figure 1, and redundant codes can be retired legitimately. A common repository, such as the new ASC implementation of sourceforge, will assist the collaborations within the LANL ASC-wide DE.

The prioritization, targets, and decisions for creating or harvesting and converting components will be recommended by peer-review and affirmed by ASC management.

References

- [1] Craig Larman, “Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process,” 2nd ed., Prentice Hall PTR, Upper Saddle River, NJ, 2002.
- [2] George T. Heineman and William T. Councill, “Component-Based Software Engineering: Putting the Pieces Together,” Addison-Wesley, 2001.

TJU:tju

Distribution:

William J. Feiereisen, CCS-DO, MS B297
Stephen R. Lee, CCS-DO, MS B297
Donald G. Shirk, CCS-DO, MS B297
Douglas B. Kothe, CCS-2, MS D413
Robert B. Lowrie, CCS-2, MS D413
Gordon L. Olson, CCS-4, MS D409
Todd J. Urbatsch, CCS-4, MS D409
Randal S. Baker, CCS-4, MS D409
Paul F. Batcho, CCS-4, MS D409
Kent G. Budge, CCS-4, MS D409
Michael W. Buksas, CCS-4, MS D409
David B. Carrington, CCS-4, MS D409
Bradley A. Clark, CCS-4, MS D409
Jon A. Dahl, CCS-4, MS D409
Thomas M. Evans, CCS-4, MS D409
Jim E. Morel, CCS-4, MS D409
Kelly G. Thompson, CCS-4, MS D409
Scott A. Turner, CCS-4, MS D409
James S. Warsa, CCS-4, MS D409
Charles W. Nakhleh, X-2, MS T085
John H. Cerutti, X-8, MS F645
CCS-4 Files